Repair of Boolean Programs with an Application to C

Andreas Griesmayer[1], Roderick Bloem[1], and Byron Cook[2]

[1] Graz University of Technology
[2] Microsoft Research

Abstract. We show how to find and fix faults in Boolean programs by extending the program to a game. In the game, the protagonist can select an alternative implementation for an incorrect statement. If the protagonist can do so successfully using a memoryless strategy that does not depend on the stack contents, we have found a correction for the Boolean program. We present a symbolic algorithm that localizes possibly faulty statements and provides corrections.

If the Boolean program is an abstraction of a C program, the repair for the Boolean program suggests a repair for the original C program. This yields a correct but incomplete approach to repairing C programs. We have applied this approach to Boolean programs that are produced as abstractions by SLAM and have thus successfully patched several faulty Windows device drivers.

1 Introduction

When a software model checker disproves a property, it typically returns a counterexample. A counterexample, however, is just an example of a failure, only a hint to the root cause of the program's error. In order to fix the bug we must understand the counterexample, find its root cause, and then implement a fix. In this paper we describe a method to automatically suggest repairs to source code based on the abstractions computed within a software model checker. With our method the programmer can either simply implement one of the proposed repairs, or the programmer may find that the proposed repairs lead to greater understanding of the root cause of the counterexample returned by the software model checker.

The technical contribution of our work is a method to fix faulty Boolean programs by computing a memoryless, stackless strategy. We assume that the software model checker is based on predicate abstraction [11] to Boolean programs [5] (i.e., pushdown automata), as is done in tools like SLAM [3]. The Boolean program can be converted into a game between the system (protagonist) and the environment (antagonist). Given a suspect expression, the system decides how the expression should behave, whereas the environment resolves nondeterminism. Such a game can be regarded as the pushdown equivalent of a reactive module [1]. A winning strategy for this game is one that ensures that the specification is adhered to by fixing the proper decisions for the system. If such a strategy exists, we can fix the Boolean program by implementing the decisions that the strategy prescribes. We are looking for a repair that changes the program as little as possible, so that it remains amenable to further modification by the programmer. The repair should depend only on global variables and the local variables that are currently in scope, and not on the stack contents. A change that does not satisfy these constraints cannot be easily implemented in a Boolean program. Such a repair corresponds to a memoryless, stackless strategy [2]. Our choice to repair the program by replacing expressions works relatively well in practice, but is not the only possibility. Our technical approach is applicable to different fault models as well.

By replacing the Boolean predicates by the expressions they represent, we obtain a constraint describing a set of repairs for the original C program. Any such repair leads to a C program that adheres to its specification. The programmer then selects a repair that does not violate any implicit assumptions. Although a repair for the Boolean program guarantees the existence of a repair for the C program, the converse does not hold. The abstract program may contain spurious counterexamples as well as the real counterexample that corresponds to the bug and it may not be possible to repair all of these. In order to demonstrate the viability of our approach we have implemented the proposed method and used it to compute suggested repairs for several Microsoft Windows device drivers after analysis using SLAM [3].

Related Work. The work described here extends work done in [13, 17] on locating and correcting faults in finite-state systems.

Alur, La Torre, and Madhusudan [2] give a fixpoint computation algorithm for solving modular pushdown games that is similar to the algorithm we present in this paper. They do not apply it to repair and do not show an implementation. They further focus on complexity analysis for reachability in different settings of visibility: global memory, local memory and local but persistent memory.

Work by Walukiewicz [18] focuses on computing strategies for more general μ-calculus properties on pushdown systems. This work is not in the setting of repair and the strategies that are found are not in general memoryless. Basing a repair on them would significantly alter the program by adding a second concurrent thread. Bouajjani, Esparza, and Maler [6] give algorithms for reachability analysis in alternating pushdown systems. Their algorithm is polynomial, but the strategy it produces is an alternating automaton and may depend on the contents of the stack.

There has been considerable work in fault localization. Most of it is of heuristic nature and relies on similarities between incorrect traces of the program and their differences with similar, correct traces [4, 12, 19]. Unlike our approach, this work requires the existence of correct executions that are similar to the counterexamples found, but most importantly, none of this work addresses repair. The approach of [4], for instance, marks as suspect the statements that appear in different failure traces but not in traces that satisfy the specification, and thus works on the basic block level. There is no guarantee that the statements found can be used to repair the program or that a possible repair location is found, even if it exists. Nevertheless, the approach appears to be quite good at finding faulty statements. It would be interesting to see if it can be used as a preprocessing step to our algorithm to limit the number of statements that we attempt to repair.

In [8], fault localization is extended to abstract counterexamples. The authors argue that explanations of abstract counterexamples are more informative than explanations of concrete counterexamples, because the predicates used capture the important information in the program, but not more: it is an automatically generated high-level description of the program. For instance, the information that x should be greater than y may be much more informative that the information that x should be 8239 and not 4.

Demsky and Rinard [9] and Khurshid, García, and Suen [14] present work on repairing corrupt data structures without terminating the program. Their work is on recovering from a failure, not on fixing faults in the system.

2 Boolean Programs

2.1 Syntax and Semantics

Boolean programs [5] are similar to C programs: they have functions and recursion, global and local variables. The difference is that all variables are of Boolean domain and no additional storage is available. Boolean programs also support assertions, parallel assignments, and nondetermism. In the following, we give a short formalization. We will not give the details of the execution model, but it can easily be defined in terms of a virtual machine. Note that a Boolean program may have more than one execution, depending on nondeterminism.

A *Boolean program* is a tuple $(R, \mathtt{main}, V_g)$, where R is a set of routines, $\mathtt{main} \in R$ is the *initial routine*, and V_g is a set of *global variables*. A *routine* $r \in R$ is a tuple (S_r, V_r), where $S_r = (s_{r,0}, \dots, s_{r,f})$ is a sequence of statements and V_r is a set of local variables. Statement $s_{r,0}$ is the *initial statement* and $s_{r,f}$ is the *final statement*.

The set of variables visible in r is $V_r' = V_g \cup V_r$. A valuation $\xi \subseteq V_r'$ is the subset of the visible variables that is set to 1, and the set of valuations in routine r is $X_r = 2^{V_r'}$.

We will not define the statements in detail. For a formalization it suffices to define two functions: The control flow graph is given by $\mathit{next}(\xi, s, s')$ meaning that control may continue at $s' \in S_r$ after executing $s \in S_r$ with valuation $\xi \in X_r$. For conditional statements, s' depends on ξ. (Because of nondeterminism, a conditional statement may have multiple successors for one ξ.) In particular, if $s_{r,i}$ is a function call, it is followed by the statement $s_{r,i+1}$, not by the first statement of the called routine.

The change of the valuation that results from executing $s \in S_r$ is denoted by $\tau_s \subseteq X_r \times X_r$. For instance, if s assigns 1 to variable a, then $\tau_{(s)} = \{(\xi, \xi \cup \{a\}) \mid \xi \in X_r\}$. The expression `choose[v,w]` expresses nondeterminism. It evaluates to 1 if v is 1. Otherwise, it evaluates to 0 if w is 1, and nondeterministically otherwise. Thus, if statement s is `a := choose[v,w]` then $\tau_s = \big\{(\xi, \xi') \mid \xi \setminus \{a\} = \xi' \setminus \{a\} \wedge (v \in \xi \rightarrow a \in \xi') \wedge ((v \notin \xi \wedge w \in \xi) \rightarrow a \notin \xi')\big\}$. Since Boolean programs are usually abstractions of C programs, and the Boolean variables are predicates, not all valuations are possible. Nondeterminism can be limited to feasible valuations by an *enforce* statement. We will not take such statements into account in the formalization, but they are easily added and are handled by our implementation.

The set of *states* of a routine is $Q_r = S_r \times X_r$ and the set of *initial states* is $I_r = \{s_{r,0}\} \times X_r$. The set of states of a program is the (disjoint) union of the set of states of its routines. The initial states of the program are the initial states of `main`. A state (s, ξ) is a *bad state* if s is an assert statement and ξ is a valuation that violates the assertion.

For a call statement s from routine *src* to routine *dst*, we use a relation $\mu_s : X_{src} \times X_{dst}$. This mapping handles the assignment of the actual parameters to the formal parameters. The values of global variables remain unchanged and local variables that are not formal parameters are assigned nondeterministically. We use the function ρ_s :

$X_{src} \times X_{dst} \to X_{src}$ to compute the valuation after the call returns. It copies the values of the global variables from the called function, copies the values of the local variables from the values they had before the call, and assigns the value returned by dst.

Example. We give an example of a Boolean program in Fig. 1(a). It was generated by abstraction of the C program in Fig. 2, using the predicates shown in Fig. 3. We only have global variables and $V'_{\mathtt{main}} = V'_{\mathtt{f}} = \{p_1, p_2, p_3\}$, so $X_{\mathtt{main}} = X_{\mathtt{f}} = X = 2^{\{p_1,p_2,p_3\}}$. The first statement in Line 1 is a parallel assignment which simultaneously assigns values to all variables. Thus, $\tau_1 = \{(\xi, \{p_2, p_3\}) \mid \xi \in X\}$. (The subscript to τ refers to the line number.) Note that `f` does not have arguments or return results. Thus, $\mu_2 = \mu_8 = \{(\xi, \xi) \mid \xi \in X\}$ and $\rho_2(\xi_{\mathtt{main}}, \xi_{\mathtt{f}}) = \rho_8(\xi_{\mathtt{main}}, \xi_{\mathtt{f}}) = \xi_{\mathtt{f}}$.

Routine `f` assigns new values to all of the variables in Line 6 and calls itself until `p2` = 0. In Line 3 of the main routine, an assertion checks that $p_1 = 1$. The program does not fulfill this requirement. □

2.2 Model Checking of Boolean Programs

We review model checking of Boolean programs [5, 10]. Given a Boolean program $P = (R, \mathtt{main}, V_g)$, we associate with every routine $r \in R$ an *execution graph* $\mathcal{E}_r = \langle Q_r, E_r \rangle$, where Q_r is the set of states of r and $E_r \subseteq Q_r \times Q_r$ is a set of edges (to be defined). Intuitively, an edge represents a step in the execution and elements of E^* represent executions. ((The relation E^* is closely related to the *path edges* used in interprocedural dataflow analysis [15].)) We associate with P an execution graph $\mathcal{E} = \langle Q, E \rangle$, where $Q = \bigcup_{r \in R} Q_r$, $E = E_c \cup \bigcup_{r \in R} E_r$. The set E_c of *call edges* is defined below.

The definitions of the edge relations E_r and E_c are mutually recursive. First, we define an update relation τ'_s on the statements. If s is a statement other than a call, then $\tau'_s = \tau_s$. Otherwise τ'_s complies with the called routine dst: $\tau'_s = \big\{(\xi_1, \xi_2) \mid \exists \xi'_1, \xi'_2 : (\xi_1, \xi'_1) \in \mu_s \wedge ((s_{dst,0}, \xi'_1), (s_{dst,f}, \xi'_2)) \in E^*_{dst} \wedge \rho_s(\xi_1, \xi'_2) = \xi_2\big\}$.

Furthermore, we define the *reachable states Rch* and the sets of edges E_r and E_c as follows.

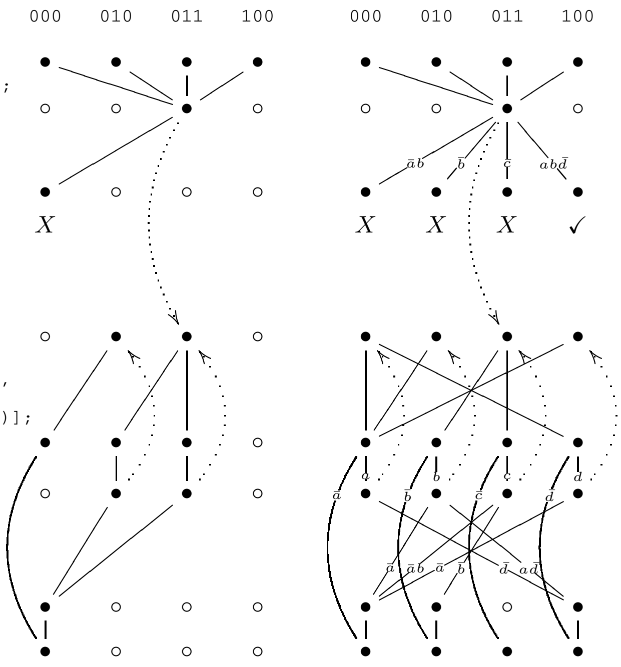
$$\begin{aligned} \mathit{Rch} &= \big\{(s, \xi) \mid \exists \xi_0 : ((s_{\mathtt{main},0}, \xi_0), (s, \xi)) \in E^*\big\}, \\ E_r &= \big\{((s, \xi), (s', \xi')) \mid (s, \xi) \in \mathit{Rch} \wedge \mathit{next}(\xi, s, s') \wedge (\xi, \xi') \in \tau'_s\big\}, \text{ and} \\ E_c &= \big\{((s_{src}, \xi), (s_{dst,0}, \xi')) \mid s_{src} \text{ calls } dst \wedge (s_{src}, \xi) \in \mathit{Rch} \wedge (\xi, \xi') \in \mu_{s_{src}}\big\}. \end{aligned}$$

Thus, we start with the initial states of the program and add edges from states as they become reachable. In particular, edges from call statements are added as soon as paths through the called routine are calculated.

Theorem 1. *For $q, q' \in Q$, we have $(q, q') \in E^*$ iff there is an execution of the program that reaches q and subsequently reaches q'.*

Example. Fig. 1(b) gives the execution graph of the example. An example of an execution starts in Line 1 with $p_1 = p_2 = p_3 = 0$. (We denoted this state by $(1, (0, 0, 0))$.) Then it progresses to state $(2, (0, 1, 1))$ and calls `f` with the same valuation. Then, the

```
decl p1, p2, p3;
0 main()
1  p1, p2, p3 := 0, 1, 1;
2  f ();
3  assert(p1);
4 end;
5 f ()
6  p1, p2, p3:=
     choose[0,(p2|p3|p1)],
     p3,
     choose[0,(!p2|!p3|p1)];
7  if(p2)    \\if(*sys)
8   f ();
9  fi
10 end
```



(a) Boolean Program (b) Execution Graph (c) Game Graph

Fig. 1. Boolean program and corresponding execution and game graph. There is one column for every consistent valuation. The valuation is given as a triplet $p_1p_2p_3$. Reached states are given by •, unreachable states by ◦. Control flow goes from the top to the bottom, except for the dotted lines, which denote call edges.

```
1 static int x;

2 void main() {
3   x = 3;
4   f ();
5   assert(x == 0);
  }
6 void f () {
7   x = x - 1;
8   if (x > 1) {
9     f ();
    }
  }
```

Fig. 2. Faulty C program

p1: x == 0
p2: x > 1
p3: x > 2

Fig. 3. Predicates used for abstraction

a: $*_{\text{sys}}(000) = 1$	$\bar{a}$: $*_{\text{sys}}(000) = 0$
b: $*_{\text{sys}}(010) = 1$	$\bar{b}$: $*_{\text{sys}}(010) = 0$
c: $*_{\text{sys}}(011) = 1$	$\bar{c}$: $*_{\text{sys}}(011) = 0$
d: $*_{\text{sys}}(100) = 1$	$\bar{d}$: $*_{\text{sys}}(100) = 0$

Fig. 4. Abbreviations for the conditions on the implementation of $*_{\text{sys}}$

valuation nondeterministically becomes 010 or 011. It then recurs and, unless the recursion is infinite, finishes `f` with valuation 000. It returns to main in that state, showing that the assertion can be violated. □

3 Repair of Boolean Programs

Suppose a Boolean program contains executions that violate an assert statement. A *repair* is a replacement of an existing statement such that the resulting program can not reach a bad state. Such a replacement can involve the `choose` function, and thus be nondeterministic. In this case, the program does not reach a bad state for any behavior of the nondeterministic statement.

To find repairs, a proper *fault model* has to be used. In the following, we assume that the program contains one fault, viz. an incorrect expression. This is only one possible fault model. Our algorithm is independent of the used fault model and other models could be used, including changes to the left hand side of an assignment or insertion or deletion of statements at arbitrary positions. We can thus adjust our approach to different areas of application, but this is not in the scope of this paper.

3.1 Building the Game

To compute a correct replacement for an existing expression, we extend the model checking algorithm to compute games between the environment (the antagonist) and the system (the protagonist). We extend the set of expressions with the construct $*_{\text{sys}}$. This construct represents an arbitrary function controlled by the system. The environment controls nondeterminism through the `choose` function.

We replace a given expression by $*_{\text{sys}}$ and ask the following question: is there an implementation for $*_{\text{sys}}$ such that the bad states are avoided regardless of the choices of the environment? We do this by negating the question and computing under which implementations of $*_{\text{sys}}$ the bad states are reached. If we then replace $*_{\text{sys}}$ by any *other* implementation, the resulting Boolean program satisfies its specification. We allow the implementation of $*_{\text{sys}}$ to be an arbitrary expression in terms of the visible variables. Thus, $*_{\text{sys}}$ does not introduce extra memory, nor can the result of $*_{\text{sys}}$ depend on the content of the stack. Computation of $*_{\text{sys}}$ corresponds to the computation of a modular strategy in [2]. In Section 3.5 we show how to attempt repair on all expressions simultaneously.

3.2 Computing the Strategy

Let P be a Boolean program $(R, \texttt{main}, V_g)$ that contains exactly one occurrence of $*_{\text{sys}}$. In order to compute a proper implementation of $*_{\text{sys}}$, we define the game graph of P. In this graph, we may progress from one state to another under one implementation of $*_{\text{sys}}$ and not under another. This holds both for statements in which $*_{\text{sys}}$ occurs and for call statements, because the behavior of the called routine may depend on the implementation of $*_{\text{sys}}$.

Suppose that $*_{\text{sys}}$ occurs in routine r. A set of valuations $c \subseteq X_r$ defines an implementation of $*_{\text{sys}}$: for $\xi \in X_r$ we have $*_{\text{sys}}(\xi) = 1$ if and only if $\xi \in c$. Let $C = 2^{X_r}$ be the set of possible implementations.

We associate with every routine $r \in R$ a *game graph* $\mathcal{G}_r = \langle Q_r, E_r \rangle$, where Q_r is the set of states of r and $E_r \subseteq Q_r \times C \times Q_r$ is a set of labeled edges (to be defined). The game graph of P is $\mathcal{G} = \langle Q, E \rangle$, where $Q = \bigcup_{r \in R} Q_r$, $E = E_c \cup \bigcup_{r \in R} E_r$. ($E_c \subseteq Q \times S \times Q$ is defined below.) We define the reflexive transitive closure of a labeled edge relation $E \subseteq Q \times C \times Q$ as $E^* = \{(q, c, q'') \mid q = q'' \vee \exists q' : (q, c, q') \in E^* \wedge (q', c, q'') \in E\}$.

For game graphs, $\tau'_s \subseteq X_r \times C \times X_r$ takes into account the implementation of $*_{\text{sys}}$. If s is a statement that does not include $*_{\text{sys}}$ and is not a call, we have $\tau'_s = \{(\xi, c, \xi') \mid c \in C \wedge (\xi, \xi') \in \tau_s\}$. We will use an example to show how $*_{\text{sys}}$ is handled. Given an implementation c, the statement $s = (a := *_{\text{sys}})$, assigns 1 to a in states ξ with $\xi \in c$. Thus, we have

$$\tau'_s = \{(\xi, c, \xi') \mid c \in C \wedge (\xi \in c \wedge \xi' = \xi \cup \{a\}) \vee (\xi \notin c \wedge \xi' = \xi \setminus \{a\}).$$

(If the set of valuations is limited using an enforce statement, we do not include edges to impossible valuations.) Finally, if s is a call to dst,

$$\begin{aligned}\tau'_s = \{(\xi_1, c, \xi_2) \mid \exists \xi'_1, \xi'_2 :\\ (\xi_1, \xi'_1) \in \mu_s, ((s_{dst,0}, \xi'_1), c, (s_{dst,f}, \xi'_2)) \in E^*_{dst} \wedge \rho_s(\xi_1, \xi'_2) = \xi_2\}.\end{aligned}$$

Now we can define

$$\begin{aligned}
R_c &= \big\{(s, \xi) \mid \exists \xi_0 : ((s_{\texttt{main},0}, \xi_0), c, (s, \xi)) \in E^*\big\},\\
E_r &= \big\{((s, \xi), c, (s', \xi')) \mid (s, \xi) \in R_c \wedge next(\xi, s, s') \wedge (\xi, c, \xi') \in \tau'_s\big\},\\
E_c &= \big\{((s_{src}, \xi), c, (s_{dst,0}, \xi')) \mid s_{src} \text{ calls } dst \wedge (s_{src}, \xi) \in R_c \wedge (\xi, \xi') \in \mu_{s_{src}}\big\}.
\end{aligned}$$

We have the following lemma.

Lemma 1. *For $c \in C$, let P'_c be the Boolean program in which $*_{sys}$ is replaced by the function described by c. There is an edge $(q, c, q') \in E^*$ iff P'_c contains an execution that reaches q and subsequently reaches q'.*

Symbolic Computation. Iterating over all possible implementation of $*_{\text{sys}}$ is very inefficient: there are 2^{X_r} such implementations. The algorithm can be implemented symbolically using binary decision diagrams (BDDs) [7]. For each valuation $\xi \in X_r$ we introduce a BDD variable x_ξ that we refer to as a *condition*: $x_\xi = 1$ (0) iff $*_{\text{sys}}(\xi) = 1$ (0, resp.). Thus, an assignment to the variables corresponds to an implementation $c \subseteq 2^{X_r}$. We construct a BDD for τ'_s for every s and use these BDDs to construct E^* symbolically. The algorithm is analogous to the explicit one, except that we handle all possible implementations simultaneously. The number of BDD variables is exponential in the number of variables in scope, which is the major bottleneck for efficiency. Thus, our algorithm is doubly exponential in the number of variables and exponential in the number of nodes in the graph, which matches the lower bound shown in [2].

Example. We attempt to repair the Boolean program by replacing the if statement in Line 7 by $*_{\text{sys}}$. (See Fig. 1(c) for the game graph.) The edges between the states are labeled with a Boolean function over X_r that represents all implementations for that edge. The conditions are given in Fig. 4. For instance, b represents all implementations such that $*_{\text{sys}}(0, 1, 0) = 1$.

Consider state $(6, (0, 1, 1))$ of routine `f`. From here, an execution may nondeterministically proceed to $(7, (0, 1, 0))$ and then to Line 10, provided that $*_{\text{sys}}(0, 1, 0) = 0$. Because the other edges on the path do not restrict the implementation of $*_{\text{sys}}$, the program can proceed, from state $(6, (0, 1, 1))$ to $(10, (0, 1, 0))$ if $*_{\text{sys}}(0, 1, 0) = 0$. This condition propagates to Line 2 of `main`, which calls `f`. Thus, in main, the program proceeds from $(2, (0, 1, 1))$ to $(3, (0, 1, 0))$ if $*_{\text{sys}}(0, 1, 0) = 0$, and then violates the assertion. Thus, a successful implementation of $*_{\text{sys}}$ cannot have $*_{\text{sys}}(0, 1, 0) = 0$. □

3.3 Extracting a Repair

To extract a repair from the game graph, we select all paths $(v, c, v') \in E^*_{main}$ that connect an initial state with a bad state. Implementations that allow these paths are faulty. The set of correct implementations is thus $I = C \setminus \bigcup\{c \mid \exists(x, c, y) \in E^* \wedge x \in I_{main} \wedge y \in \mathit{bad}\}$, where bad is the set of bad states.

Theorem 2. *If we replace $*_{sys}$ with any implementation $c \in I$, the resulting program contains no execution that leads to a bad state.*

If I is given symbolically, each prime implicant of I corresponds to a repair. The BDD variables V_P that appear positively in the implicant denote the conditions under which the implementation must return 1 and the negative variables V_N denote conditions under which the implementation must return 0. Thus, $*_{\text{sys}}$ must return 1 (0) for implementations ξ such that $x_\xi \in V_P$ ($x_\xi \in V_N$, resp.) Identifying variables with the conditions they denote, the repair can be given as a set of statements `choose[`$\bigvee_{x \in V_P} x, \bigvee_{x \in V_N} x$`]`, one for each (irredundant) prime implicant.

Example. Consider again Fig. 1(c). There are paths from the initial states to error states for any implementation that satisfies $\bar{a}b \vee \bar{b} \vee \bar{c}$, which can be simplified to $\bar{a} \vee \bar{b} \vee \bar{c}$. Thus, $I = abc$. Therefore, the suggested repair is `choose`$(a \vee b \vee c, 0)$, or, in terms of predicates, $\mathit{choose}(\neg p1 \wedge (p2 \vee \neg p3), 0)$. □

3.4 Mapping Boolean Repairs to C

Suppose the Boolean program is a conservative abstraction of the C program. A repair for a Boolean program corresponds to a repair for the C program. If we substitute the meaning of the predicates in the repair for the Boolean program, we obtain a constraint for the C program. This constraint requires that in a given line a given predicate becomes true in some situations and false in others. Any implementation that satisfies this constraint is guaranteed satisfy the specification. Note that there may be more than one implementation that satisfies the constraint. It is up to the programmer to select a good repair, depending on the intended semantics.

Example. Recall that the Boolean program that we have repaired is an abstraction of the C program in Fig. 2. The meaning of the predicates p_1, p_2, and p_3 is given in Fig. 3. The repair that we have found says that the then branch should certainly be taken if $x \neq 0 \wedge (x > 1 \vee x \leq 2)$, which is equivalent to $x \neq 0$. Substituting $x \neq 0$ in Line 8 gives the correct behavior. Note that the suggestion allows us to take the then branch also in other cases. In particular, we could satisfy it by substituting true in Line 8. Although this satisfies the specification (the assertion is never violated), it is clearly undesirable, as it introduces an infinite loop. In general, user interaction is necessary to select the desired repair from the set of possible repairs given by the algorithm. We have observed, however, that the suggested repairs are typically quite good. They reduce the number of statements to be considered to just a few and give good hints on how to modify the statements. A more realistic example is found in the next section. □

3.5 Localizing Faults

With the graphs we defined so far, we can localize the fault by successively replacing every expression in the program by $*_{\text{sys}}$ and computing the related game graph. Expressions for which a repair is found are potential fault locations. These games differ only in the implementation of one expression. To avoid redundant computations, we compute a combined game graph for all possible repairs at one time. To this end, we compute τ' for each statement once with the original expression, and a second time for its implementation with $*_{\text{sys}}$. The combined graph is built using labels identifying the different subgraphs for the repair of each statement. Using symbolic representation, this can be done quite efficiently.

4 Experimental Results

In order to demonstrate the viability of our approach, we have performed experiments on Boolean programs constructed by the SLAM-based Static Driver Verifier [3] , using our implementation based on CUDD [16]. We examined nine bugs in drivers from the Windows operating system. We used a representative set of drivers that implemented various functionalities, including storage, input, networking, etc. For this reason we believe that the results should be repeatable for other drivers and other code of similar size and complexity. The driver's code size ranged from 2,000 to 35,000 lines of C code. (Between 1,700 and 25,000 lines in the Boolean program.)

For verification, a driver is accompanied by a test harness and an automaton. The harness is the same in every test and contains a routine that nondeterministically calls functions in the drivers API. The automaton contains code to test if a given property holds.

Table 1 lists the results of the experiments. This table contains the following information: (1) the name of the driver, (2) the number of lines of code of the Boolean program, (3) the number of expressions examined for repair, (4) the number of repairs in the Boolean program, (5) the number of repairs in the C-code of the driver (all other repairs attempt to fix the bug by changing the test harness or the automaton), (6) the overall run time, (7) number of global variables and the maximum number of local variables, (8) the results of the approach (discussed below), and (9) the name of the tested property. Table 2 contains brief and informal descriptions of the properties that we used.

We examined the repairs computed for the Boolean program and checked if they can be used for the C code of the driver. Examples where the real fault was within the set of found repairs are marked with $\checkmark$. For the remaining examples we either found

Table 1. Results from experiments with Boolean programs produced by SLAM when checking properties of Windows device drivers

Driver	LoC	# Expr.	# Total	# in Driver	Time(s)	# vars	Results	Property
1394 diag	7223	273	57	8	1345	2/10	✓	MarkIrpPending
bulltlp3.1	4751	860	30	3	16482	13/15	X^1	IrpProcComplete
daytona	14364	305	2	0	379	2/0	X^1	StartIoRecursion
gameenum	4001	217	29	1	577	2/9	✓	MarkIrpPending
hidgame	3611	335	27	4	7132	9/17	X^2	LowerDriverReturn
mousefilter	1755	165	21	3	4035	7/33	✓	PendCompleteReq
parport	24379	1055	3	1	8334	2/0	✓	DoubleCompletion
pscr	4842	374	5	0	2797	6/7	X^1	IrqlReturn
sfloppy	2216	19	6	4	4	2/0	✓	AddDevice

Table 2. Informal summary of properties listed in Table 1

Property	Summary
AddDevice	Checks that a driver's AddDevice routine calls certain key APIs.
DoubleCompletion	Checks that drivers do not complete I/O request packets twice.
IrpProcComplete	Ensures that dispatch routines completely process I/O request packets.
IrqlReturn	Checks that a driver dispatch routine's thread priority is the same at function call and exit.
LowerDriverReturn	Checks that if a driver calls another driver lower in stack, the dispatch routine will return the same status as lower driver.
MarkIrpPending	Ensures that returns of STATUS_PENDING and calls to IoMarkIrpPending are correlated.
PendCompleteReq	Checks that drivers do not return STATUS_PENDING if IoCompleteRequest has been called during the execution of the dispatch routine.
StartIoRecursion	Checks for potential recursion in a driver's StartIo routines.

no repairs, or the approach suggested only "cheating" repairs such as avoiding a call to the erroneous routine or staying in a loop forever. Missing the repair can have two reasons:X^1 marks examples where the fault was a missing call or the wrong order of calls. These faults do not fit in our fault model and thus cannot be found. X^2 marks examples where the abstraction was too coarse to find a repair.

In the rest of the section, we describe adjustments to our approach that made our approach feasible for real device drivers and we present a case study of the Windows parallel port driver.

4.1 Adjustments for Checking Windows Device Drivers

Limiting the Number of Variables Considered for Repair: While the examples contain up to 40 predicates visible at one time, many of them are temporary local variables that are uninteresting. Global variables hold information from the test harness, which give informations about the driver's environment. By reducing the number of variables considered for the implementation of $*_{\mathrm{sys}}$, we may drastically reduce the size of the BDD. On the other hand, we can miss results that are not expressible with the limited set. No incorrect repairs are generated. For five of the examples, this heuristic was necessary to be able to compute repairs.

Parallel Assignments: Parallel assignments are hard to handle: Repairing all expressions at the same time is very inefficient. Repairing only one is usually infeasible because the assignments are tightly related and because many valuations of the predicates

```
state { bool CompletionAlreadyCalled = 0; }

IoCompleteRequest.entry
{
  if (SdvHarnessIrp==$1) {
    if (CompletionAlreadyCalled) { error(); }
    else { CompletionAlreadyCalled = 1; }
  }
}
```

Fig. 5. Temporal property *DoubleCompletion*

```
133 NTSTATUS
134 PptDispatchClose(PDEVICE_OBJECT DevObj,PIRP Irp) {
135    PFDO_EXTENSION fdx = DevObj->DeviceExtension;
136    P5TraceIrpArrival( DevObj, Irp );
137    if( DevTypeFdo == fdx->DevType ) {
138       return PptFdoClose( DevObj, Irp );
139    } else {
140       return PptPdoClose( DevObj, Irp );
141 }  }
```

Fig. 6. Source code from dispatchRedirect.c in Parallel port device driver

are ruled out using an enforce statement. Therefore, we searched for a new implementation of one expression, while allowing the other predicates to take arbitrary values in accordance with the enforce rule. The repairs for *1394 diag* and *gameenum* suggest new values in parallel assignments which were not possible without this optimization.

Removing Nondeterministic Functions: The abstraction refinement process starts with a nondeterministic program and adds predicates when unfeasible paths are reported. In some cases, a feasible counterexample is found before any predicates for some of the functions are discovered. Such functions induce infeasible paths which make it impossible to repair the Boolean program. In two cases, *bulltlp3.1* and *hidgame*, we removed calls to such functions from the harness in order to find repairs. In contrast to the other heuristics, this one can produce repairs which are not valid in the original C program. Tighter integration of the approach with SLAM would instead trigger further abstraction in such cases.

4.2 Case Study: Windows Parallel Port Device Driver

We will now describe how we have used our approach to find a repair for a buggy Windows parallel port device driver. The relevant code is given in Figs. 6, 7, and 8. The code in Figure 5 describes a temporal property, *DoubleCompletion*. The driver violates this property, which ensures that the device driver dispatch routines do not call the kernel-level API function IoCompleteRequest more than once on the same I/O request packet. (This example originally appears in [3]) The Windows kernel function IoCompleteRequest frees up the space of a request packet, which may then be re-allocated and passed to another thread in the system. Calling IoCompleteRequest twice with the same parameter can have disastrous consequences to the system's stability.

When trying to prove that the device driver does not violate the rule, the test harness calls the device driver's dispatch routines nondeterministically, using an I/O request packet called SdvHarnessIrp. The erroneous execution is as follows. The test harness calls the parallel port device driver's *close* dispatch routine PptDispatchClose (Fig. 6, Line 134) on the I/O request packet SdvHarnessIrp. The function IoCompleteRequest will erroneously be called twice on this package. When PptFdoClose (Fig. 7, Line 4) is called, we enter the conditional statement at Line 14 and call P4CompleteRequest (Fig. 8, Line 1774). Then, in Line 1782, IoCompleteRequest is called on SdvHarnessIrp and the function returns to the call site. We then leave the conditional statement via the goto on Line 18. At Line 60 P4CompleteRequestReleaseRemLock (Fig. 8, Line 1786) is called, which itself calls P4CompleteRequest and thus makes the second call to IoCompleteRequest on SdvHarnessIrp.

```
04 NTSTATUS PptFdoClose(
05   IN  PDEVICE_OBJECT  DeviceObject,
06   IN  PIRP            Irp
07 ) {
08   PFDO_EXTENSION fdx=DeviceObject->DeviceExtension;
09   NTSTATUS       status;
10
11   PAGED_CODE();
12
13   // Verify that device was not SUPRISE_REMOVED.
14   if(fdx->PnpState & PPT_DEVICE_SURPRISE_REMOVED) {
15     // Our device has been SURPRISE removed, but
16     // since this is a CLOSE, SUCCEED anyway
17     status=P4CompleteRequest(Irp,STATUS_SUCCESS,0);
18     goto target_exit;
19   }
...
59  target_exit:
60   DD((PCE)fdx,DDT,"PptFdoClose - ........");
61   return P4CompleteRequestReleaseRemLock(
62           Irp, STATUS_SUCCESS, 0, &fdx->RemoveLock);
63 }
```

Fig. 7. Source code from fdoClose.c in Parallel port device driver

```
1774 NTSTATUS P4CompleteRequest(
1775   IN PIRP       Irp,
1776   IN NTSTATUS   Status,
1777   IN ULONG_PTR  Information
1778 ){
1779   P5TraceIrpCompletion(Irp);
1780   Irp->IoStatus.Status      = Status;
1781   Irp->IoStatus.Information = Information;
1782   IoCompleteRequest(Irp,IO_NO_INCREMENT);
1783   return Status;
1784 }
1785
1786 NTSTATUS P4CompleteRequestReleaseRemLock(
1787   IN PIRP              Irp,
1788   IN NTSTATUS          Status,
1789   IN ULONG_PTR         Information,
1790   IN PIO_REMOVE_LOCK  RemLock
1791 ) {
1792   P4CompleteRequest(Irp,Status,Information);
1793   PptReleaseRemoveLock(RemLock,Irp);
1794   return Status;
1795 }
```

Fig. 8. Source code from util.c in Parallel port device driver

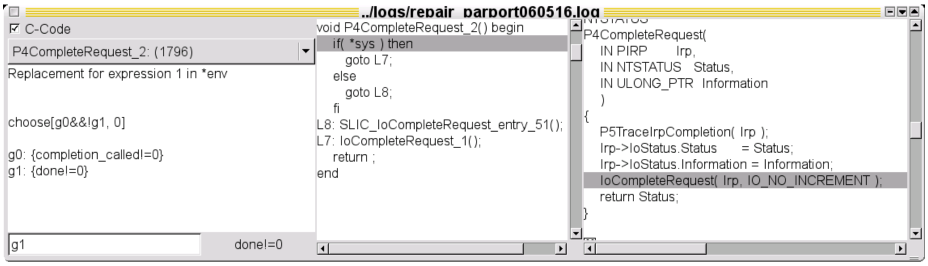
```
 1 void P4CompleteRequest_2() begin
 2     if( *sys ) then
 3        goto L7;
 4     else
 5        goto L8;
 6     fi
 7 L8: SLIC_IoCompleteRequest_entry_51();
 8 L7: IoCompleteRequest_1();
 9     return ;
10 end
```

Fig. 9. Boolean routine from abstraction of P4-CompleteRequest (Fig. 8, line 1775)

```
  if (SdvHarnessIrp==R) {
    if (CompletionAlreadyCalled) {}
    else {
      CompletionAlreadyCalled = 1;
      IoCompleteRequest(R);
    }
  }
```

Fig. 10. Replacement for the call of Io-CompleteRequest in P4CompleteRequest

Our algorithm finds three repairs for the Boolean program, including Line 2 of routine P4CompleteRequest_2, shown in Fig. 9, which is the abstraction of the routine P4CompleteRequest. The abstraction of the driver contains two predicates: g0: CompletionAlreadyCalled $\neq 0$ and g1: done $\neq 0$. We know the predicate CompletionAlreadyCalled from the safety property. Variable done is defined by the SDV harness and is true iff we are still verifying the property (Irp is still equal to SdvHarnessIrp). The if statement in Line 2 decides whether or not to call SLIC_IoCompleteRequest_entry_51 which executed an abstracted version of the rule-checking code in Fig. 5. The original



../logs/repair_parport060516.log

☑ C-Code

P4CompleteRequest_2: (1796)

Replacement for expression 1 in *env

choose[g0&&!g1, 0]

g0: {completion_called!=0}
g1: {done!=0}

g1   done!=0

```
void P4CompleteRequest_2() begin
   if( *sys ) then
      goto L7;
   else
      goto L8;
   fi
L8: SLIC_IoCompleteRequest_entry_51();
L7: IoCompleteRequest_1();
   return ;
end
```

```
P4CompleteRequest(
   IN PIRP       Irp,
   IN NTSTATUS   Status,
   IN ULONG_PTR  Information
   )
{
   P5TraceIrpCompletion( Irp );
   Irp->IoStatus.Status      = Status;
   Irp->IoStatus.Information = Information;
   IoCompleteRequest( Irp, IO_NO_INCREMENT );
   return Status;
}
```

Fig. 11. Tool for viewing the repairs. On the left, we can choose which repair to examine. The text fields give details on the repair and the corresponding parts of the Boolean program and C code.

routine, IoCompleteRequest_1(), is called in any case. The suggested repair for Line 2 is `if(choose[g0&&!g1,0])`, which means we may not enter IoCompleteRequest if we are still checking the property and CompletionAlreadyCalled is 0. A screen shot of the tool used to examine the repairs is given in Fig.11.

A possible implementation of this result in the driver is to add a new variable CompletionAlreadyCalled that is initialized to zero (as in the automaton) and a variable SdvHarnessIrp which is initialized to the value of the I/O request packet passed into the driver dispatch routine. Thus, we repair the driver by replacing the call to IoCompleteRequest in P4CompleteRequest by the code given in Fig. 10. Note that the repair of the Boolean program is memoryless because the automaton that implements the temporal property is included in the Boolean program. For the C program this is not the case, and we need to add a variable.

5 Conclusions

Modern software model checkers provide counterexamples when they disprove a property. While clearly useful, counterexamples are not what the programmer is eventually wants: *a correct program*. In this paper we have presented a method to automatically suggest repairs in Boolean program. Given a model checker for C that uses Boolean programs as an abstraction, we can use this method to fix faults in C programs. If a repair is found for the Boolean program, then there is a repair for the C program. Our approach often yields useful results, as shown by the application of our algorithm to buggy Windows device drivers.

Future research includes finding fixes for violations of liveness constraints such as infinite recursion. It would also be interesting to integrate repair and refinement more tightly, by adding new predicates when needed for repair. Our approach is inefficient when there are many Boolean variables in scope at the same time. One way to speed the algorithm up may be a preprocessing step by a fault localization tool to narrow down the set of suspect statements. Finally, our fault model may not be ideal. Instead of replacing existing expressions, we may need to consider other repairs, such as the insertion of statements. The theory presented here works regardless of the fault model.

Acknowledgments

The authors would like to thank Ranjit Jhala and the anonymous reviewers for their valuable comments. This work was supported in part by the European Commission under contract 507219 (PROSYD).

References

[1] R. Alur and T. A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15:7–48, 1999.

[2] R. Alur, S. La Torre, and P. Madhusudan. Modular strategies for recursive game graphs. In *Tools and Algorithms for the Construction and the Analysis of Systems (TACAS'03)*, pages 363–378, 2003.

[3] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *European Systems Conference (EuroSys'06)*, 2006.

[4] T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: Localizing errors in counterexample traces. In *30th Symposium on Principles of Programming Languages (POPL 2003)*, pages 97–105, 2003.

[5] T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for Boolean programs. In *SPIN 00: SPIN Workshop*, pages 113–130. 2000.

[6] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model checking. In *Proc. 8th Int. Conf. on Concurrency Theory (CONCUR'97)*, pages 135–150. 1997.

[7] R. E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24:293–318, 1992.

[8] S. Chaki, A. Groce, and O. Strichman. Explaining abstract counterexamples. In *Proc. of the International Symposium on Foundations of Software Engineering*, pages 73–82, 2004.

[9] B. Demsky and M. Rinard. Automatic detection and repair of errors in data structures. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'03)*, pages 78–95, 2003.

[10] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *Twelfth Conference on Computer Aided Verification (CAV'00)*, pages 324–336. Springer-Verlag, 2000.

[11] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Ninth Conference on Computer Aided Verification (CAV'97)*, pages 72–83. 1997.

[12] A. Groce. Error explanation with distance metrics. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS'04)*, pages 108–122, 2004.

[13] B. Jobstmann, A. Griesmayer, and R. Bloem. Program repair as a game. In *17th Conference on Computer Aided Verification (CAV'05)*, pages 226–238 2005.

[14] S. Khurshid, I. García, and Y. Suen. Repairing structurally complex data. In *SPIN Workshop on Model Checking of Software (SPIN'05)*, pages 123–138, 2005.

[15] T. Reps, S. Horwitz, and S. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Symposium on Principles of Programming Languages*, pages 49–61, 1995.

[16] F. Somenzi. *CUDD: CU Decision Diagram Package*. University of Colorado at Boulder, ftp://vlsi.colorado.edu/pub/.

[17] S. Staber, B. Jobstmann, and R. Bloem. Finding and fixing faults. In *13th Conference on Correct Hardware Design and Verification Methods (CHARME '05)*, pages 35–49. 2005.

[18] I. Walukiewicz. Pushdown processes: Games and model-checking. *Information and Computation*, 157:234–263, 2000.

[19] A. Zeller. Isolating cause-effect chains from computer programs. In *10th Int. Symp. on the Foundations of Software Engineering (FSE-10)*, pages 1–10, November 2002.